Parallel programming with OpenACC directives

Alexei Strelchenko

CaSToRC @ The Cyprus Institute

GPU workshop, December 12th, 2012

Outline

- Challenges of heterogeneous programming
- What is OpenACC
- First example
- OpenACC Programming Model
- Tips and hints
- Summary

Challenges of heterogeneous programming

- Levels of parallelism:
  - multiple device with separate (physical) address spaces
  - multiple threads inside a device
  - vector parallelism inside the thread
- Types of parallel programming models:
  - Message Passing (distributed memory systems)
  - Task parallel (shared memory systems, thread based)
  - Data parallel, streaming processing
- Exploiting heterogeneous environment:
  - GPGPU APIs and programming languages (OpenCL, CUDA)
  - Libraries (CUBLAS, CUSPARSE, MAGMA etc)
  - Compiler directives (PGI Accelerator Directives, OpenACC)

Programming with OpenACC directives

- Developed by NVIDIA, PGI, Cray and CASP
- Provides:
  - Compiler directives to specify parallel regions (in C/Fortran)
  - Implicit accelerator initialization
  - Implicit data transfers between host and accelerator
  - Interoperability with CUDA C/Fortran and GPU libs
- Almost as simple as OpenMP

OpenACC: how it looks like

- OpenMP:

```
int main() {
  double pi = 0.0;
#pragma omp parallel for reduction(+:pi)
  for(long i=0; i<N; i++){
    double t = (double)((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
  }
}
```

- OpenACC code

```
int main() {
  double pi = 0.0;
#pragma acc kernels
  for(long i=0; i<N; i++){
    double t = (double)((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
  }
}
```

OpenACC directive syntax

- C:

#pragma acc *directive [clause [,] clause] ...]*

parallel code block

- Fortran:

!$acc *directive [clause [,] clause] ...]*

parallel code block

!$acc end *directive*

OpenACC main constructs

- Parallel and kernel regions
- Parallel loops
- Data regions
- Runtime API

Kernels constructs

- Each loop compiled into a separate GPU kernel
- C:

```
#pragma acc kernels
{
  for(int i=0; i<N; i++){
  ....
  }//end of loop
  for(int j=0; j<N; j++){
  ....
  }//end of loop
}
```

- Fortran:

```
!$acc kernels
do i=0,i<N
   ....
end do
do j=0,j<N
   ....
end do
!$acc end kernels
```

Kernels construct syntax

- C:

```
#pragma acc kernels [clause ...]
{
  structured block
}
```

- Fortran:

```
!$acc kernels [clause ...]
  structured block
!$acc end kernels
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - data management clauses

Loop construct syntax

- C:

```
#pragma acc loop [clause ...]
{
  for (...)
    ...
}
```

- Fortran:

```
!$acc loop [clause ...]
  do ...
    ...
  end do
!$acc end loop
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - data management clauses, execution clauses

SXPY example

- C:

```
void sxpy (float *x, float *y, int n){
#pragma acc parallel loop independent
   for (int i < 0; i < n; i++)
     y[i] += x[i];
}
```

- Fortran:

```
subroutine sxpy (x, y, n)
integer :: n, i
real :: x(:), y(:)
!$acc parallel loop
   do i = 1, n
     y(i) = x(i) + y(i)
   end do
!$acc end parallel
end subroutine
```

- *independent* clause used in loop directive:

limits the effect of pointer aliasing (for C)

SXPY example

- The restrict keyword

```
#include <openacc.h>
void sxpy (float *x, float *restrict y, int n){
#pragma acc kernels
   for (int i < 0; i < n; i++)
     y[i] += x[i];
}
```

- float *restrict ptr :

alternative to *independent* clause: no aliasing assumed for ptr

Code compilation with PGI

- C:

  pgcc -acc [-Minfo=accel] -ta=nvidia, host -o sxpy sxpy.c

- Fortran:

  pgf90 -acc [-Minfo=accel] -ta=nvidia, host -o sxpy sxpy.f90

- Options:
  - -acc: enables OpenACC recognition
  - -Minfo=accel: enables compiler feedback
  - -ta=nvidia, cc1x cc2x cc3x: sets target architecture
  - -ta=nvidia, cuda4.x: sets cuda toolkit version
  - -ta=nvidia, fastmath O3: device code optimizations

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Compiler output

```
pgcc -acc -Minfo=accel -ta=nvidia, host -o sxpy sxpy.c

sxpy:
     13, Generating copy(y[0:len])
         Generating copyin(x[0:len])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     14, Loop is parallelizable
         Accelerator kernel generated
         14, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 48 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 60 constant, 0 local memory bytes
```

Toy model: 2D Poisson equation

The 2D Poisson equation:

$$\nabla^2 u(x,y) := u_{xx} + u_{yy} = f(x,y),\ \ x,y \in \Omega$$

Here: $f(x,y)$ - known function, $u(x,y)$ - solution, and $u_x := \partial_x u$

Boundary conditions:

- Dirichlet: $u(x,y) = g(x)$ on boundary $\partial\Omega$
- Neumann $\frac{\partial u}{\partial \mathbf{n}} = g(x)$ on boundary $\partial\Omega$
- Mixed etc.

2D Poisson equation: Jacobi iterations

- Finite difference approximation for the equation:

$$\frac{u_{j-1,i} - 2u_{j,i} + u_{j+1,i}}{\Delta_y^2} + \frac{u_{j,i-1} - 2u_{j,i} + u_{j,i+1}}{\Delta_x^2} = f_{i,j}$$

Here: $\Delta_x = x_{i+1} - x_i,\ 0 < i < NX - 1$
$\Delta_y = y_{j+1} - y_j,\ 0 < j < NY - 1$

- The algorithm (Dirichlet BC):

Update the solution until reach desired accuracy ($\Delta_y = \Delta_x$):

$$u_{j,i}^{new} = \frac{1}{4}(u_{j-1,i} + u_{j+1,i} + u_{j,i-1} + u_{j,i+1}) + F_{j,i},\ \ F_{j,i} = \frac{\Delta_x^2 f_{j,i}}{4}$$

On the boundary $\partial\Omega$ we impose $u_{j,i}^{new} = u_{j,i}$
Convergence criterio: $\max_\Omega |u_{j,i}^{new} - u_{j,i}| < \epsilon$
Number of iterations scales as $\max(NX^2, NY^2)\ \ln(\epsilon)$

2D Poisson equation: Jacobi iterations

Input – initial guess U, scaled F, and temporal buffer:

$U[ny][nx]$,$F[ny][nx]$, $W[ny][nx]$

Set tolerance ϵ, iteration accuracy: $\delta = 1.0$
Start Jacobi iterations:
While $\delta > \epsilon$ **do**:
  For $j = 1, .., ny - 1$ **do**:
    For $i = 1, .., nx - 1$ **do**:
      Calculate 5pt stencil:
      $W[j][i] = F[j][i] + 0.25(U[j][i+1] + U[j][i-1] + U[j+1][i] + U[j-1][i])$
      Update error $\delta = \max(\delta, |W[j][i] - U[j][i]|)$
    End of i-loop
  End of j-loop

  Update solution for interior grid points:
  $U \leftarrow W$

End

2D Poisson: Jacobi iterations with OpenMP

Input – initial guess U, scaled F, and temporal buffer:

$U[ny][nx]$,$F[ny][nx]$, $W[ny][nx]$

Set tolerance ϵ, iteration accuracy: $\delta = 1.0$
Start Jacobi iterations:
While $\delta > \epsilon$ **do**:
#pragma omp parallel for shared(ny, nx, W, U, F)
For $j = 1, .., ny - 1$ **do**:
For $i = 1, .., nx - 1$ **do**:
Calculate 5pt stencil:
$W[j][i] = F[j][i] + 0.25(U[j][i+1] + U[j][i-1] + U[j+1][i] + U[j-1][i])$
Update error $\delta = \max(\delta, |W[j][i] - U[j][i]|)$
End of i-loop
End of j-loop

Update solution for interior grid points:
$U \leftarrow W$

End

2D Poisson: Jacobi iterations with OpenACC

Start Jacobi iterations:

While $\delta > \epsilon$ **do**:

#pragma acc kernels

{

For $j = 1, .., ny - 1$ **do**:

For $i = 1, .., nx - 1$ **do**:

Calculate 5pt stencil:

$W[j][i] = F[j][i] + 0.25(U[j][i+1] + U[j][i-1] + U[j+1][i] + U[j-1][i])$

Update error $\delta = \max(\delta, |W[j][i] - U[j][i]|)$

End of i-loop

End of j-loop

Update solution for interior grid points:

For $j = 1, .., ny - 1$ **do**:

For $i = 1, .., nx - 1$ **do**:

$U[j][i] = W[j][i]$

End of i-loop

End of j-loop

} //end of acc kernels region

End

Data construct

- C:

```
#pragma acc data [clause ...]
{
  structured block
}
```

- Fortran:

```
!$acc data [clause ...]
  structured block
!$acc end data
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - data management clauses (*copy()*, *create()* etc.)

Data clauses

- copy(list):

Allocates memory on GPU and copies data H2D when entering region and copies data D2H when exiting the region

- copyin(list):

Allocates memory on GPU and copies data H2D when entering region

- copyout(list):

Allocates memory on GPU and copies data D2H when exiting the region

- create(list):

Allocates memory on GPU, no data transfers

- present(list):

Data is already present on GPU from another containing data region

Data clauses (continued)

- present_or_copy(list)
- present_or_copyin(list)
- present_or_copyout(list)
- present_or_create(list)

Shortcut versions for the above: pcopy(list), pcopyin(list) etc.

Data clauses: array shaping

- Data clauses can be used on *data*, *kernels* or *parallel* directives:
- C:

```
#pragma acc data copy(a[n:len])
{
  structured block
}
#pragma acc kernels copyin(b[0:n])
{
  structured block
}
```

- Fortran:

```
!$acc data copy(a[n:len])
  structured block
!$acc end data

!$acc kernels copyin(b[1:n])
  structured block
!$acc end kernels
```

2D Poisson: OpenACC data region

Start Jacobi iterations:

```
#pragma acc data copy(U[0:ny][0:nx]), copyin(F[1:ny-2][1:nx-2]),
                  create(W[1:ny-2][1:nx-2])
```

While $\delta > \epsilon$ **do**:

```
#pragma acc kernels
```

{

For $j = 1, .., ny - 1$ **do**:

For $i = 1, .., nx - 1$ **do**:

Calculate 5pt stencil:

$W[j][i] = F[j][i] + 0.25(U[j][i+1] + U[j][i-1] + U[j+1][i] + U[j-1][i])$

Update error $\delta = max(\delta, |W[j][i] - U[j][i]|)$

End of i-loop

End of j-loop

Update solution for interior grid points:

$U \leftarrow W$

} //end of acc kernels region

End do

Update Construct

- C:

```
#pragma acc update [clause ...]
{
  structured block
}
```

- Clauses:
  - if (*condition*)
  - async (*expression*)
  - host(list)
  - device(list)

- Fortran:

```
!$acc update [clause ...]
  structured block
!$acc end data
```

General tips

- Nested loops are best for parallelization
- Iterations must be independent (restrict keyword/independent clause)
- Compiler must be able to figure out sizes of data region
- Use contiguous memory for multi-dimensional arrays
- Function calls must be inlineable (-Minline)
- Conditional compilation with _OPENACC macro

Reduction clause

- Reduction clause is allowed on *parallel* and *loop* constructs:
  - C:

```
#pragma acc parallel reduction(operation : var)
{
  structured block with reduction on var
}
```

  - Fortran:

```
!$acc parallel reduction(operation : var)
  structured block with reduction on var
!$acc end kernels
```

2D Poisson: OpenACC reduction

Start Jacobi iterations:

#pragma acc data copy(U[0:ny][0:nx]), copyin(F[1:ny-2][1:nx-2]),
create(W[1:ny-2][1:nx-2])

While $\delta > \epsilon$ **do**:

#pragma acc parallel loop reduction(max, err)

For $j = 1, .., ny - 1$ **do**:

For $i = 1, .., nx - 1$ **do**:

Calculate 5pt stencil:

$W[j][i] = F[j][i] + 0.25(U[j][i+1] + U[j][i-1] + U[j+1][i] + U[j-1][i])$

Update error $\delta = max(\delta, |W[j][i] - U[j][i]|)$

End of i-loop

End of j-loop

#pragma acc kernels loop pcopyin(U[1:ny-2][1:nx-2]), pcreate(W[1:ny-2][1:nx-2])

Update solution for interior grid points:

$U \leftarrow W$

End

Combined directives

- *parallel* and *kernels* can be combined with *loop* directive:
  - C:

```
#pragma acc parallel loop
{
  for(..)
}
```

  - Fortran:

```
!$acc parallel loop
  do ...
  end do
!$acc end loop
```

OpenACC execution model

- OpenACC exe model three levels: *gang*, *worker* and *vector*
- Allows to better fit a particular target architecture
- On CPU : *gang*, *worker* and *vector*
  - gang(*number*) :number of CPUs in the system
  - worker(*number*) :number of CPU cores
  - vector(*number*) : e.g. SIMD instructions (SSE, AVX)
- On GPU : *gang* and *vector*
  - gang(*number*) :number of thread blocks per grid
  - vector(*number*) :number of threads per block

Execution model examples

- 64 thread blocks, 64 threads per block:

($N = 128 \times 64$, each thread executes 2 loop instructions)

```
#pragma acc kernels loop gang(64) vector(64)
  for(int i = 0; i < N; i++) y[i] += x[i];
```

- 128 thread blocks, 128 threads per block:

```
#pragma acc parallel num_gangs(128) vector_length(64)
{
#pragma acc loop
  for(int i = 0; i < N; i++) y[i] += x[i];
}
```

- Multi-dimensional exe domain

(128×256 blocks, each block of 32×64 threads)

```
#pragma acc kernels loop gang(256) vector(64)
  for(int j = 0; j < N; j++)
#pragma acc loop gang(128) vector(32)
    for(int i = 0; i < M; i++) ...
```

2D Poisson: explicit multi-dimensional execution

Start Jacobi iterations:

#pragma acc data copy(U[0:ny][0:nx]), copyin(F[1:ny-2][1:nx-2]),
create(W[1:ny-2][1:nx-2])

While $\delta > \epsilon$ **do**:

#pragma acc kernels loop gang(16) vector(32)

For $j = 1, .., ny - 1$ **do**:

#pragma acc loop gang(16) vector(32)

For $i = 1, .., nx - 1$ **do**:

Calculate 5pt stencil:

$W[j][i] = F[j][i] + 0.25(U[j][i+1] + U[j][i-1] + U[j+1][i] + U[j-1][i])$

Update error $\delta = max(\delta, |W[j][i] - U[j][i]|)$

End of i-loop

End of j-loop

#pragma acc loop collapse(2)

Update solution for interior grid points (two for-loops!):

$U \leftarrow W$

End

OpenACC RT API

- System setup routines
  - acc_init(*acc_device_nvidia*)
  - acc_set_device_type(*acc_device_nvidia*)
  - acc_set_device_num(dev_id, *acc_device_nvidia*)
- Resource allocation routines:
  - void* acc_malloc(size_t);
  - void acc_free(void*)
- Synchronization routines:
  - acc_async_wait(int);
  - acc_async_wait_all()

Interoperability with CUDA: deviceptr clause

- deviceptr(*list*):

Declare that the ptrs in *list* refer to device ptrs that need not be allocated or moved between host and device

- Example:

```
#include <openacc.h>
cudaMalloc (&x, n*sizeof(float));
cudaMalloc (&y, n*sizeof(float));
  ....
#pragma acc data deviceptr(x, y)
{
#pragma acc kernels
    for (int i < 0; i < n; i++)
      y[i] += x[i];
}
```

Interoperability with CUDA: host_data construct

Makes the address of device data available on the host

- use_device(*list*):

Use the device address for any variable in the list

- Example:

```
#include <openacc.h>
#pragma acc data copy(x[0:n])
{
#pragma acc host_data use_device(x)
    use_cuda_ptr(x);
}
```

OpenACC with OpenMP

- Multi-GPU execution with OpenMP:

```
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(number_of_gpus)
{
    int id = omp_get_thread_num();
    acc_set_device_num(id, acc_device_nvidia );
#pragma acc parallel
    {
    //Do something on GPU id;
    }
}
```

OpenACC with MPI

- Multi-GPU execution with MPI:

```
#include <openacc.h>
#include <mpi.h>

  int my_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, & my_rank);
  int num_dev = acc_get_num_devices(acc_device_nvidia);
  int id = my_rank % num_dev;
  acc_set_device_num(id, acc_device_nvidia );
#pragma acc parallel
    {
    //Do something on GPU id;
    }
```

OpenACC future perspectives

- Integration into OpenMP (promised in 2013)
- More tools (GNU)
- More languages (C++)
- More targets: CPUs, AMD GPUs, Intel MIC (will be provided by PGI in 2013)
- Interoperability with OpenCL
- Multiple GPUs

Thank you!